ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

STANDARD ECMA-55

Minimal BASIC

January 1978

Free copies of this **ECMA** standard are available from
ECMA European Computer Manufacturers Association
114 Rue du Rhône - 1204 Geneva (Switzerland)

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

STANDARD ECMA-55

Minimal BASIC

January 1978

BRIEF HISTORY

The first version of the language BASIC, acronym for Beginner's All-purpose Symbolic Instruction Code, was produced in 1964 at the Dartmouth College in the USA. This version of the language was oriented towards interactive use. Subsequently, a number of implementations of the language were prepared, that differed in part from the original one.

In 1974, the ECMA General Assembly recognized the need for a standardized version of the language, and in September 1974 the first meeting of the ECMA Committee TC 21, BASIC, took place. In January 1974, a corresponding committee, X3J2, had been founded in the USA.

Through a strict co-operation it was possible to maintain full compatibility between the ANSI and ECMA draft standards. The ANSI one was distributed for public comments in January 1976, and a number of comments were presented by ECMA.

A final version of the ECMA Standard was prepared at the meeting of June 1977 and adopted by the General Assembly of ECMA on Dec. 14, 1977 as Standard ECMA-55.

TABLE OF CONTENTS

		Page
1.	SCOPE	1
2.	REFERENCES	1
3.	DEFINITIONS	1
	3.1 BASIC	1
	3.2 Batch Mode	1
	3.3 End-of-Line	2
	3.4 Error	2
	3.5 Exception	2
	3.6 Identifier	2
	3.7 Interactive Mode	2
	3.8 Keyword	2
	3.9 Line	2
	3.10 Nesting	2
	3.11 Print Zone	3
	3.12 Rounding	3
	3.13 Significant Digits	3
	3.14 Truncation	3
4.	CHARACTERS AND STRINGS	3
5.	PROGRAMS	5
6.	CONSTANTS	6
7.	VARIABLES	8
8.	EXPRESSIONS	9
9.	IMPLEMENTATION SUPPLIED FUNCTIONS	11
10.	USER DEFINED FUNCTIONS	13
11.	LET STATEMENT	14
12.	CONTROL STATEMENTS	15
13.	FOR AND NEXT STATEMENTS	17
14.	PRINT STATEMENT	19
15.	INPUT STATEMENT	21
16.	READ AND RESTORE STATEMENTS	23
17.	DATA STATEMENT	24
18.	ARRAY DECLARATIONS	25
19.	REMARK STATEMENT	26
20.	RANDOMIZE STATEMENT	26
TABLE 1 - BASIC Character Set		28
TABLE 2 - BASIC Code		29
APPENDIX 1 -	Organization of the Standard	30
APPENDIX 2 -	Method of Syntax Specification	32
APPENDIX 3 -	Conformance	34
APPENDIX 4 -	Implementation-defined Features	35

1. SCOPE

This Standard ECMA-55 is designed to promote the interchangeability of BASIC programs among a variety of automatic data processing systems. Subsequent Standards for the same purpose will describe extensions and enhancements to this Standard. Programs conforming to this Standard, as opposed to extensions or enhancements of this Standard, will be said to be written in "Minimal BASIC".

This Standard establishes:

- the syntax of a program written in Minimal BASIC.
- The formats of data and the precision and range of numeric representations which are acceptable as input to an automatic data processing system being controlled by a program written in Minimal BASIC.
- The formats of data and the precision and range of numeric representations which can be generated as output by an automatic data processing system being controlled by a program written in Minimal BASIC.
- The semantic rules for interpreting the meaning of a program written in Minimal BASIC.
- The errors and exceptional circumstances which shall be detected and also the manner in which such errors and exceptional circumstances shall be handled.

Although the BASIC language was originally designed primarily for interactive use, this Standard describes a language that is not so restricted.

The organization of the Standard is outlined in Appendix 1. The method of syntax specification used is explained in Appendix 2.

2. REFERENCES

ECMA-6 : 7-Bit Input/Output Coded Character Set, 4th Edition

ECMA-53 : Representation of Source Programs

3. DEFINITIONS

For the purposes of this Standard, the following terms have the meanings indicated.

3.1 BASIC

A term applied as a name to members of a special class of languages which possess similar syntaxes and semantic meanings; acronym for Beginner's All-purpose Symbolic Instruction Code.

3.2 Batch-mode

The processing of programs in an environment where no provision is made for user interaction.

3.3 End-of-line

The character(s) or indicator which identifies the termination of a line. Lines of three kinds may be identified in Minimal BASIC: program lines, print lines and input reply lines. End-of-line may vary between the three cases and may also vary depending upon context. Thus, for example, an end of input line may vary on a given system depending on the terminal being used in interactive or batch mode.

Typical examples of end-of-line are carriage-return, carriage-return line-feed, and end of record (such as end of card).

3.4 Error

A flaw in the syntax of a program which causes the program to be incorrect.

3.5 Exception

A circumstance arising in the course of executing a program which results from faulty data or computations or from exceeding some resource constraint. Where indicated certain exceptions (non-fatal exceptions) may be handled by the specified procedures; if no procedure is given (fatal exceptions) or if restrictions imposed by the hardware or operating environment make it impossible to follow the given procedure, then the exception shall be handled by terminating the program.

3.6 Identifier

A character string used to name a variable or a function.

3.7 Interactive mode

The processing of programs in an environment which permits the user to respond directly to the actions of individual programs and to control the commencement and termination of these programs.

3.8 Keyword

A character string, usually with the spelling of a commonly used or mnemonic word, which provides a distinctive identification of a statement or a component of a statement of a programming language.

The keywords in Minimal BASIC are: BASE, DATA, DEF, DIM, END, FOR, GO, GOSUB, GOTO, IF, INPUT, LET, NEXT, ON, OPTION, PRINT, RANDOMIZE, READ, REM, RESTORE, RETURN, STEP, STOP, SUB, THEN and TO.

3.9 Line

A single transmission of characters which terminates with an end-of-line.

3.10 Nesting

A set of statements is nested within another set of statements when:

- the nested set is physically contiguous, and
- the nesting set (divided by the nested set) is non-null.

3.11 Print zone

A contiguous set of character positions in a printed output line which may contain an evaluated print statement element.

3.12 Rounding

The process by which the representation of a value with lower precision is generated from a representation of higher precision taking into account the value of that portion of the original number which is to be omitted.

3.13 Significant digits

The contiguous sequence of digits between the high-order non-zero digit and the low-order non-zero digit, without regard for the location of the radix point. Commonly, in a normalized floating point internal representation, only the significant digits of a representation are maintained in the significance.

NOTE: The Standard requires that the ability of a conforming implementation to accept numeric representations be measured in terms of significant digits rather than the actual number of digits (that is including leading or trailing zeroes) in the representation.

3.14 Truncation

The process by which the representation of a value with lower precision is generated from a representation of higher precision by merely deleting the unwanted low order digits of the original representation.

4. CHARACTERS AND STRINGS

4.1 General Description

The character set for BASIC is contained in the ECMA 7-bit coded character set. Strings are sequences of characters and are used in BASIC programs as comments (see 19), as string constants (see 6), or as data (see 15).

4.2 Syntax

1. letter	=	A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z
2. digit	=	0/1/2/3/4/5/6/7/8/9
3. string-character	=	quotation-mark / quoted-string-character
4. quoted-string-character	=	exclamation-mark / number-sign / dollar-sign / percent-sign / ampersand / apostrophe / left-parenthesis / right-parenthesis / asterisk / comma / solidus / colon / semi-colon / less-than-sign / equals-sign / greater-than-sign / question-mark / circumflex-accent / underline / unquoted-string-character
5. unquoted-string-character	=	space / plain-string-character

6. plain-string-character = plus-sign / minus-sign / full-stop / digit / letter
7. remark-string = string-character*
8. quoted-string = quotation-mark quoted-string-character* quotation-mark
9. unquoted-string = plain-string-character / plain-string-character unquoted-string-character* plain-string-character

4.3 Examples

ANY CHARACTERS AT ALL (?!*!!) CAN BE USED IN A "REMARK".
"SPACES AND COMMAS CAN OCCUR IN QUOTED STRINGS."
COMMAS CANNOT OCCUR IN UNQUOTED STRINGS.

4.4 Semantics

The letters shall be the set of upper-case Roman letters contained in the ECMA 7-bit coded character set in positions 4/1 to 5/10.

The digits shall be the set of arabic digits contained in the ECMA 7-bit coded character set in positions 3/0 to 3/9.

The remaining string-characters shall correspond to the remaining graphic characters in position 2/0 to 2/15, 3/10 to 3/15 and in positions 5/14, 5/15 of the ECMA 7-bit coded character set.

The names of characters are specified in Table 1.

The coding of characters is specified in Table 2; however, this coding applies only when programs and/or input/output data are exchanged by means of coded media.

4.5 Exceptions

None.

4.6 Remarks

Other characters from the ECMA 7-bit coded character set (including control characters) may be accepted by an implementation and may have a meaning to some other processor (such as an editor) but have no prescribed meaning within this Standard. Programs containing characters other than the string-characters described above are not standard-conforming programs.

The several kinds of characters and strings described by the syntax correspond to the various uses of strings in a BASIC program. Remark-strings may be used in remark-statements (see 19). Quoted-strings may be used as string-constants (see 6). Unquoted-strings may be used in addition to quoted-strings as data elements (see 17) without being enclosed in quotation marks; unquoted-strings cannot contain leading or trailing spaces.

5. PROGRAMS

5.1 General Description

BASIC is a line-oriented language. A BASIC program is a sequence of lines, the last of which shall be an end-line and each of which contains a keyword. Each line shall contain a unique line-number which serves as a label for the statement contained in that line.

5.2 Syntax

```
1. program          = block* end-line
2. block            = (line/for-block)*
3. line             = line-number statement end-of-line
4. line-number      = digit digit? digit? digit?
5. end-of-line      = [implementation-defined]
6. end-line         = line-number end-statement end-of-line
7. end-statement    = END
8. statement        = data-statement / def-statement /
                      dimension -statement / gosub-statement /
                      goto-statement / if-then-statement /
                      input-statement / let-statement /
                      on-goto-statement / option-statement /
                      print-statement / randomize-statement /
                      read-statement / remark-statement /
                      restore-statement / return-statement /
                      stop-statement
```

5.3 Examples

```
999 END
```

5.4 Semantics

A BASIC program shall be composed of a sequence of lines ordered by line-numbers, the last of which contains an end-statement. Program lines shall be executed in sequential order, starting with the first line, until

- some other action is dictated by a control statement, or
- an exception condition occurs, which results in a termination of the program, or
- a stop-statement or end-statement is executed.

Special conventions shall be observed regarding spaces. With the following exceptions, spaces may occur anywhere in a BASIC program without affecting the execution of that program and may be used to improve the appearance and readability of the program.

Spaces shall not appear:

- at the beginning of a line
- within keywords
- within numeric constants
- within line numbers
- within function or variable names
- within two-character relation symbols

All keywords in a program shall be preceded by at least one space and, if not at the end of a line, shall be followed by at least one space.

Each line shall begin with a line-number. The values of the integers represented by the line-numbers shall be positive nonzero; leading zeroes shall have no effect. Statements shall occur in ascending line-number order.

The manner in which the end of a statement line is detected is determined by the implementation; e.g. the end-of-line may be a carriage-return character, a carriage-return character followed by a line-feed character, or the end of a physical record.

Lines in a standard-conforming program may contain up to 72 characters; the end-of-line indicator is not included within this 72 character limit.

The end-statement serves both to mark the physical end of the main body of a program and to terminate the execution of the program when encountered.

5.5 Exceptions

None.

5.6 Remarks

Local editing facilities may allow for the entry of statement lines in any order and also allow for duplicate line-numbers and lines containing only a line-number. Such editing facilities usually sort the program into the proper order and in the case of duplicate line-numbers, the last line entered with that line-number is retained. In many implementations, a line containing only a line-number (without trailing spaces) is usually deleted from the program.

6. CONSTANTS

6.1 General Description

Constants can denote both scalar numeric values and string values.

A numeric-constant is a decimal representation in positional notation of a number. There are four general syntactic forms of (optionally signed) numeric constants:

- implicit point representation	sd...d
- explicit point unscaled representation	sd..drd..d
- explicit point scaled representation	sd..drd..dEsd..d
- implicit point scaled representation	sd..dEsd..d

where:

d is a decimal digit,
r is a full-stop
s is an optional sign, and
E is the explicit character E.

A string-constant is a character string enclosed in quotation marks (see 4).

6.2 Syntax

```
1. numeric-constant  = sign? numeric-rep
2. sign              = plus-sign / minus-sign
3. numeric-rep       = significand exrad?
4. significand       = integer full-stop? / integer? fraction
5. integer           = digit digit*
6. fraction          = full-stop digit digit*
7. exrad             = E sign? integer
8. string-constant   = quoted-string
```

6.3 Examples

```
1        500          -21.         .255        1E10
5E-1     .4E+1
"XYZ"              "X - 3B2"             "1E10"
```

6.4 Semantics

The value of a numeric-constant is the number represented by that constant. "E" stands for "times ten to the power"; if no sign follows the symbol "E", then a plus sign is understood. Spaces shall not occur in numeric-constants.

A program may contain numeric representations which have an arbitrary number of digits, though implementations may round the values of such representations to an implementation-defined precision of not less than six significant decimal digits. Numeric constants can also have an arbitrary number of digits in the exrad, though nonzero constants whose magnitude is outside an implementation-defined range will be treated as exceptions. The implementation-defined range shall be at least 1E-38 to 1E+38. Constants whose magnitudes are less than machine infinitesimal shall be replaced by zero, while constants whose magnitudes are larger than machine infinity shall be diagnosed as causing an overflow.

A string-constant has as its value the string of all characters between the quotation marks; spaces shall not be ignored. The length of a string-constant, i.e. the number of characters contained between the quotation-marks, is limited only by the length of a line.

6.5 Exceptions

The evaluation of a numeric constant causes an overflow (nonfatal, the recommended recovery procedure is to supply machine infinity with the appropriate sign and continue).

6.6 Remarks

Since this Standard does not require that strings with more than 18 characters be assignable to string variables (see 7), conforming programs can use string constants with more than 18 characters only as elements in a print-list.

It is recommended that implementations report constants whose magnitudes are less than machine infinitesimal as underflows and continue.

7. VARIABLES

7.1 General Description

Variables in BASIC are associated with either numeric or string values and, in the case of numeric variables, may be either simple variables or references to elements of one or two dimensional arrays; such references are called subscripted variables.

Simple numeric variables shall be named by a letter followed by an optional digit.

Subscripted numeric variables shall be named by a letter followed by one or two numeric expressions enclosed within parentheses.

String variables shall be named by a letter followed by a dollar sign.

Explicit declarations of variable types are not required; a dollar-sign serves to distinguish string from numeric variables, and the presence of a subscript distinguisnes a subscripted variable from a simple one.

7.2 Syntax

1. variable	=	numeric-variable / string-variable
2. numeric-variable	=	simple-numeric-variable / numeric-array-element
3. simple-numeric-variable	=	letter digit?
4. numeric-array-element	=	numeric-array-name subscript
5. numeric-array-name	=	letter
6. subscript	=	left-parenthesis numeric-expression (comma numeric-expression)? right-parenthesis
7. string-variable	=	letter dollar-sign

7.3 Examples

```
X          A5          V(3)          W(X,X+Y/2)
S$         C$
```

7.4 Semantics

At any instant in the execution of a program, a numeric-variable is associated with a single numeric value and a string-variable is associated with a single string value. The value associated with a variable may be changed by the execution of statements in the program.

The length of the character string associated with a string-variable can vary during the execution of a program from a length of zero characters (signifying the null or empty string) to 18 characters.

Simple-numeric-variables and string-variables are declared implicitly through their appearance in a program.

A subscripted variable refers to the element in the one or two dimensional array selected by the value(s) of the subscript(s). The value of each subscript is rounded to the nearest integer. Unless explicitly declared in a dimension statement, subscripted variables are implicitly declared by their first appearance in a program. In this case the range of each subscript is from zero to ten inclusive, unless the presence of an option-statement indicates that the range is from one to ten inclusive. Subscript expressions shall have values within the appropriate range (see 18).

The same letter shall not be the name of both a simple variable and an array, nor the name of both a one-dimensional and a two-dimensional array.

There is no relationship between a numeric-variable and a string-variable whose names agree except for the dollar-sign.

At the initiation of execution the values associated with all variables shall be implementation-defined.

7.5 Exceptions

A subscript is not in the range of the explicit or implicit dimensioning bounds (fatal).

7.6 Remarks

Since initialization of variables is not specified, and hence may vary from implementation to implementation, programs that are intended to be transportable should explicitly assign a value to each variable before any expression involving that variable is evaluated.

There are many commonly used alternatives for associating implementation-defined initial values with variables; it is recommended that all variables are recognizably undefined in the sense that an exception will result from any attempt to access the value of any variable before that variable is explicitly assigned a value.

8. EXPRESSIONS

8.1 General Description

Expressions shall be either numeric-expressions or string-expressions.

Numeric-expressions may be constructed from variables, constants, and function references using the operations of addition, subtraction, multiplication, division and involution.

String-expressions are composed of either a string-variable or a string-constant.

8.2 Syntax

1. expression = numeric-expression / string-expression
2. numeric-expression = sign? term (sign term)*
3. term = factor (multiplier factor)*
4. factor = primary (circumflex-accent primary)*
5. multiplier = asterisk / solidus
6. primary = numeric-variable / numeric-rep / numeric-function-ref / left-parenthesis numeric-expression right-parenthesis
7. numeric-function-ref = numeric-function-name argument-list?
8. numeric-function-name = numeric-defined-function / numeric-supplied-function
9. argument-list = left-parenthesis argument right-parenthesis
10. argument = numeric-expression
11. string-expression = string-variable / string-constant

8.3 Examples

```
3*X - Y^2          A(1)+A(2)+A(3)          2^(-X)
-X/Y               SQR(X^2+Y^2)
```

8.4 Semantics

The formation and evaluation of numeric-expressions follows the normal algebraic rules. The symbols circumflex-accent, asterisk, solidus, plus-sign and minus-sign represent the operations of involution, multiplication, division, addition and subtraction, respectively. Unless parentheses dictate otherwise, involutions are performed first, then multiplications and divisions, and finally additions and subtractions. In the absence of parentheses, operations of the same precedence are associated to the left.

A-B-C is interpreted as (A-B)-C, A^B^C as (A^B)^C, A/B/C as (A/B)/C and -A^B as -(A^B).

If an underflow occurs in the evaluation of a numeric expression then the value generated by the operation which resulted in the underflow shall be replaced by zero.

0^0 is defined to be 1, as in ordinary mathematical usage.

When the order of evaluation of an expression is not constrained by the use of parentheses, and if the mathematical use of operators is associative, commutative, or both, then full use of these properties may be made in order to revise the order of evaluation of the expression.

In a function reference, the number of arguments supplied shall be equal to the number of parameters required by the definition of the function.

A function reference is a notation for the invocation of a predefined algorithm, into which the argument value, if any, is substituted for the parameter (see 9 and 10) which is used in the function definition. All functions referenced in an expression shall either be implementation-supplied or be defined in a def-statement. The result of the evaluation of the function, achieved by the execution of the defining algorithm, is a scalar numeric value which replaces the function reference in the expression.

8.5 Exceptions

- Evaluation of an expression results in division by zero (nonfatal, the recommended recovery procedure is to supply machine infinity with the sign of the numerator and continue).
- Evaluation of an expression results in an overflow (nonfatal, the recommended recovery procedure is to supply machine infinity with the algebraically correct sign and continue).
- Evaluation of the operation of involution results in a negative number being raised to a non-integral power (fatal).
- Evaluation of the operation of involution results in zero being raised to a negative value (nonfatal, the recommended recovery procedure is to supply positive machine infinity and continue).

8.6 Remarks

The accuracy with which the evaluation of an expression takes place will vary from implementation to implementation. While no minimum accuracy is specified for the evaluation of numeric-expressions, it is recommended that implementations maintain at least six significant decimal digits of precision.

The method of evaluation of the operation of involution may depend upon whether or not the exponent is an integer. If it is, then the indicated number of multiplications may be performed; if it is not, then the expression may be evaluated using the LOG and EXP functions (see 9).

It is recommended that implementations report underflow as an exception and continue.

9. IMPLEMENTATION SUPPLIED FUNCTIONS

9.1 General Description

Predefined algorithms are supplied by the implementation for the evaluation of commonly used numeric functions.

9.2 Syntax

1. numeric-supplied-function = ABS / ATN / COS / EXP / INT / LOG / RND / SGN / SIN / SQR / TAN

9.3 Examples

None.

9.4 Semantics

The values of the implementation-supplied functions, as well as the number of arguments required for each function, are described below. In all cases, X stands for a numeric expression.

Function	Function value
ABS(X)	The absolute value of X.
ATN(X)	The arctangent of X in radians, i.e. the angle whose tangent is X. The range of the function is -(pi/2) < ATN(X) < (pi/2) where pi is the ratio of the circumference of a circle to its diameter.
COS(X)	The cosine of X, where X is in radians.
EXP(X)	The exponential of X, i.e. the value of the base of natural logarithms (e = 2,71828...) raised to the power X; if EXP(X) is less than machine infinitesimal, then its value shall be replaced by zero.
INT(X)	The largest integer not greater than X; e.g. INT(1.3) = 1 and INT(-1.3) = -2.
LOG(X)	The natural logarithm of X; X must be greater than zero.
RND	The next pseudo-random number in an implementation-supplied sequence of pseudo-random numbers uniformly distributed in the range 0 <= RND < 1 (see also 20).
SGN(X)	The sign of X: -1 if X < 0, 0 if X = 0 and +1 if X > 0.
SIN(X)	The sine of X, where X is in radians.
SQR(X)	The nonnegative square root of X; X must be nonnegative.
TAN(X)	The tangent of X, where X is in radians.

9.5 Exceptions

- The value of the argument of the LOG function is zero or negative (fatal).
- The value of the argument of the SQR function is negative (fatal).
- The magnitude of the value of the exponential or tangent function is larger than machine infinity (nonfatal, the recommended recovery procedure is to supply machine infinity with the appropriate sign and continue).

9.6 Remarks

The RND function in the absence of a randomize-statement (see 20) will generate the same sequence of pseudo-random numbers each time a program is run. This convention is chosen so that programs employing pseudo-random numbers can be executed several times with the same result.

It is recommended that, if the value of the exponential function is less than machine infinitesimal, implementations report this as an underflow and continue.

10. USER DEFINED FUNCTIONS

10.1 General Description

In addition to the implementation supplied functions provided for the convenience of the programmer (see 9), BASIC allows the programmer to define new functions within a program.

The general form of statements for defining functions is

```
            DEF FNx = expression
or          DEF FNx (parameter) = expression
```

where x is a single letter and a parameter is a simple numeric-variable.

10.2 Syntax

1. def-statement	=	DEF numeric-defined-function parameter-list? equals-sign numeric-expression
2. numeric-defined-function	=	FN letter
3. parameter-list	=	left-parenthesis parameter right-parenthesis
4. parameter	=	simple-numeric-variable

10.3 Examples

```
DEF FNF(X) = X^4 - 1        DEF FNP = 3.14159
DEF FNA(X) = A*X + B
```

10.4 Semantics

A function definition specifies the means of evaluating the function in terms of the value of an expression involving the parameter appearing in the parameter-list and possibly other variables or constants. When the function is referenced, i.e. when an expression involving the function is evaluated, then the expression in the argument list for the function reference, if any, is evaluated and its value is assigned to the parameter in the parameter-list for the function definition (the number of arguments shall correspond exactly to the number of parameters). The expression in the function definition is then evaluated, and this value is assigned as the value of the function.

The parameter appearing in the parameter-list of a function definition is local to that definition, i.e. it is distinct from any variable with the same name outside of the function definition. Variables which do not appear in the parameter-list are the variables of the same name outside the function definition.

A function definition shall occur in a lower numbered line than that of the first reference to the function. The expression in a def-statement is not evaluated unless the defined function is referenced.

If the execution of a program reaches a line containing a def-statement, then it shall proceed to the next line with no other effect.

A function definition may refer to other defined functions, but not to the function being defined. A function shall be defined at most once in a program.

10.5 Exceptions

None.

11. LET STATEMENT

11.1 General Description

A let-statement provides for the assignment of the value of an expression to a variable. The general syntactic form of the let-statement shall be

```
LET variable = expression
```

11.2 Syntax

1. let-statement = numeric-let-statement / string-let-statement
2. numeric-let-statement = LET numeric-variable equals-sign numeric-expression
3. string-let-statement = LET string-variable equals-sign string-expression

11.3 Examples

```
LET P = 3.14159
LET A(X,3) = SIN(X)*Y + 1

LET A$ = "ABC"
LET A$ = B$
```

11.4 Semantics

The expression is evaluated (see 8) and its value is assigned to the variable to the left of the equals sign.

11.5 Exceptions

A string datum contains too many characters (fatal).

12. CONTROL STATEMENTS

12.1 General Description

Control statements allow for the interruption of the normal sequence of execution of statements by causing execution to continue at a specified line, rather than at the one with the next higher line number.

The goto-statement

```
GO TO line-number
```

allows for an unconditional transfer.

The if-then-statement

```
IF exp1 rel exp2 THEN line-number
```

where "exp1" and "exp2" are expressions and "rel" is a relational operator, allows for a conditional transfer.

The gosub and return statements

```
GO SUB line-number
RETURN
```

allow for subroutine calls.

The on-goto-statement

```
ON expression GO TO line-number, ..., line-number
```

allows control to be transferred to a selected line.

The stop-statement

```
STOP
```

allows for program termination.

12.2 Syntax

1. goto-statement	= GO space* TO line-number
2. if-then-statement	= IF relational-expression THEN line-number
3. relational-expression	= numeric-expression relation numeric-expression / string-expression equality-relation string-expression
4. relation	= equality-relation / less-than-sign / greater-than-sign / not-less / not-greater
5. equality-relation	= equals-sign / not-equals
6. not-less	= greater-than-sign equals-sign
7. not-greater	= less-than-sign equals-sign
8. not-equals	= less-than-sign greater-than-sign
9. gosub-statement	= GO space* SUB line-number
10. return-statement	= RETURN
11. on-goto-statement	= ON numeric-expression GO space* TO line-number (comma line-number)*

12. stop-statement = STOP

12.3 Examples

```
GO TO 999                    IF X > Y+83 then 200
IF A$ <> B$ THEN 550         ON L+1 GO TO 300,400,500
```

12.4 Semantics

A goto-statement indicates that execution of the program is to be continued at the specified line-number.

If the value of the relational-expression in an if-then-statement is true, then execution of the program shall continue from the specified line-number; if the value of the relational-expression is false, then execution shall be continued in sequence, i.e. with the statement on the line following that containing the if-then-statement.

The relation "less than or equal to" shall be denoted by <=. Similarly, "greater than or equal to" shall be denoted by >=, while "not equal to" shall be denoted by <>.

The relation of equality holds between two strings if and only if the two strings have the same length and contain identical sequences of characters.

The execution of the gosub-statement and the return-statement can be described in terms of a stack of line-numbers (but may be implemented in some other fashion). Prior to execution of the first gosub-statement by the program, this stack is empty. Each time a gosub-statement is executed, the line-number of the gosub-statement is placed on top of the stack and execution of the program is continued at the line specified in the gosub-statement. Each time a return-statement is executed, the line-number on top of the stack is removed from the stack and execution of the program is continued at the line following the one with that line-number.

It is not necessary that equal numbers of gosub-statements and return-statements be executed before termination of the program.

The expression in an on-goto-statement shall be evaluated and rounded to obtain an integer, whose value is then used to select a line-number from the list following the GOTO (the line-numbers in the list are indexed from left to right, starting with 1). Execution of the program shall continue at the statement with the selected line-number.

All line-numbers in control-statements shall refer to lines in the program.

The stop-statement causes termination of the program.

12.5 Exceptions

- An attempt is made to execute a return-statement without having executed a corresponding gosub-statement (fatal).

- The integer obtained as the value of an expression in an on-goto-statement is less than one or greater than the number of line-numbers in the list (fatal).

13. FOR AND NEXT STATEMENTS

13.1 General Description

The for-statement and next-statement provide for the construction of loops. The general syntactic form of the for-statement and next-statement is

```
FOR v = initial-value TO limit STEP increment
NEXT v
```

where "v" is a simple numeric variable and the "initial-value", "limit" and "increment" are numeric expressions; the clause "STEP increment" is optional.

13.2 Syntax

1.	for-block	= for-line for-body
2.	for-body	= block next-line
3.	for-line	= line-number for-statement end-of-line
4.	next-line	= line-number next-statement end-of-line
5.	for-statement	= FOR control-variable equals-sign initial-value TO limit (STEP increment)?
6.	control-variable	= simple-numeric-variable
7.	initial-value	= numeric-expression
8.	limit	= numeric-expression
9.	increment	= numeric-expression
10.	next-statement	= NEXT control-variable

13.3 Examples

```
FOR I = 1 TO 10
NEXT I
```

```
FOR I = A TO B STEP -1
NEXT I
```

13.4 Semantics

The for-statement and the next-statement are defined in conjunction with each other. The physical sequence of statements beginning with a for-statement and continuing up to and including the first next-statement with the same control variable is termed a "for-block". For-blocks can be physically nested, i.e. one can contain another, but they shall not be interleaved, i.e. a for-block which contains a for-statement or a next-statement shall contain the entire for-block begun or ended by that statement.

Furthermore, physically nested for-blocks shall not use the same control variable.

In the absence of a STEP clause in a for-statement, the increment is assumed to be +1.

The action of the for-statement and the next-statement is defined in terms of other statements, as follows:

```
        FOR v = initial-value TO limit STEP increment
        (block)
        NEXT v
```

is equivalent to:

```
        LET own1 = limit
        LET own2 = increment
        LET v = initial-value
line1   IF (v-own1) * SGN (own2) > 0 THEN line2
        (block)
        LET v = v + own2
        GOTO line1
line2   REM continued in sequence
```

Here v is any simple-numeric-variable, own1 and own2 are variables associated with the particular for-block and not accessible to the programmer, and line1 and line2 are line-numbers associated with the particular for-block and not accessible to the programmer. The variables own1 and own2 are distinct from similar variables associated with other for-blocks. A program shall not transfer control into a for-body by any statement other than a return statement (see 12).

13.5 Exceptions

None.

13.6 Remarks

Where arithmetic is approximate (as with decimal fractions in a binary machine), the loop will be executed within the limits of machine arithmetic. No presumptions about approximate achievement of the end test are made. It is noted that in most ordinary situations where machine arithmetic is truncated (rather than rounded), such constructions as

```
        FOR X = 0 TO 1 STEP 0.1
```

will work as the user expects, even though 0.1 is not representable exactly in a binary machine. If this is indeed the case, then the construction

```
        FOR X = 1 TO 0 STEP -0.1
```

will probably not work as expected.

As specified above, the value of the control-variable upon exit from a for-block via its next-statement is the first value not used; if exit is via a control-statement, the control-variable retains the value it has when the control-statement is executed.

The variables "own1" and "own2" associated with a for-block are assigned values only upon entry to the for-block through its for-statement.

14. PRINT STATEMENT

14.1 General Description

The print-statement is designed for generation of tabular output in a consistent format.

The general syntactic form of the print-statement is

PRINT item p item p ... p item

where each item is an expression, a tab-call, or null, and each punctuation mark p is either a comma or a semi-colon.

14.2 Syntax

1. print-statement = PRINT print-list?
2. print-list = (print-item? print-separator)* print-item?
3. print-item = expression / tab-call
4. tab-call = TAB left-parenthesis numeric-expression right-parenthesis
5. print-separator = comma / semicolon

14.3 Examples

```
PRINT X                              PRINT "X EQUALS", 10
PRINT X; (Y+Z)/2                     PRINT X, Y
PRINT                                PRINT ,,,X
PRINT TAB(10); A$; "IS DONE."
```

14.4 Semantics

The execution of a print-statement generates a string of characters for transmission to an external device. This string of characters is determined by the successive evaluation of each print-item and print-separator in the print-list.

Numeric-expressions shall be evaluated to produce a string of characters consisting of a leading space if the number is positive or a leading minus-sign if the number is negative followed by the decimal representation of the absolute value of the number and a trailing space. The possible formats for the decimal representation of a number are the same as those described for numeric-constants in 6 and are used as follows.

Each implementation shall define two quantities, a significance-width d to control the number of significant decimal digits printed in numeric representations, and an exrad-width e to control the number of digits printed in the exrad component of a numeric representation. The value of d shall be at least six and the value of e shall be at least two.

Each number that can be represented exactly as an integer with d or fewer decimal digits is output using the implicit point unscaled representation.

All other numbers shall be output using either explicit point unscaled notation or explicit point scaled notation. Numbers which can be represented with d or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format shall be output using the unscaled format. For example, if d = 6, then 10^(-6) is output as .000001 and

10^(-7) is output as 1.E-7.

Numbers represented in the explicit point unscaled notation shall be output with up to d significant decimal digits and a full-stop; trailing zeroes in the fractional part may be omitted. A number with magnitude less than 1 shall be represented with no digits to the left of the full-stop. This form requires up to d+3 characters counting the sign, the full-stop and the trailing space.

Numbers represented in the explicit point scaled notation shall be output in the format

significand E sign integer

where the value x of the significand is in the range 1 <= x < 10 and is to be represented with exactly d digits of precision, and where the exrad component has one to e digits. Trailing zeroes may be omitted in the fractional part of the significand and leading zeroes may be omitted from the exrad. This form requires up to d+e+5 characters counting the two signs, the full-stop, the "E" and a trailing space.

String-expressions shall be evaluated to generate a string of characters.

The evaluation of the semicolon separator shall generate the null string, i.e. a string of zero length.

The evaluation of a tab-call or a comma separator depends upon the string of characters already generated by the current or previous print-statements. The "current line" is the (possibly empty) string of characters generated since the last end-of-line was generated. The "margin" is the number of characters, excluding the end-of-line character, that can be output on one line and is defined by the implementation. The "columnar position" of the current line is the print position that will be occupied by the next character output to that line; print positions are numbered consecutively from the left, starting with position one.

Each print-line is divided into a fixed number of print zones, where the number of zones and the length of each zone is implementation defined. All print zones, except possibly the last one on a line, shall have the same length. This length shall be at least d+e+6 characters in order to accomodate the printing of numbers in explicit point scaled notation as described above and to allow the comma separator to move the printing mechanism to the next zone as described below.

The purpose of the tab-call is to set the columnar position of the current line to the specified value prior to printing the next print-item. More precisely, the argument of the tab-call is evaluated and rounded to the nearest integer n. If n is less than one, an exception occurs. If n is greater than the margin m, then n is reduced by an integral multiple of m so that it is in the range 1 <= n <= m; i.e. n is set equal to

n - m * INT ((n-1)/m).

If the columnar position of the current line is less than or equal to n, then spaces are generated, if necessary, to set the columnar position to n; if the columnar position of the current line is greater than n, then an end-of-line is generated followed by enough spaces to set the columnar position of the new current line to n.

The evaluation of the comma-separator generates one or more spaces to set the columnar position to the beginning of the next print zone, unless the current print zone is the last on the line, in which case an end-of-line is generated.

If the print list does not end in a print-separator, then an end-of-line is generated and added to the characters generated by the evaluation of the print-list.

If the evaluation of any print-item in a print-list would cause the length of a nonempty line to exceed the margin, then an end-of-line is generated prior to the characters generated by that print-item. Subsequently, if the evaluation of a print-item generates a string whose length is greater than the margin, then end-of-lines are inserted after every m characters in the string, where m is the margin value.

14.5 Exceptions

The evaluation of a tab-call argument generates a value less than one (nonfatal: the recommended recovery procedure is to supply one and continue).

14.6 Remarks

The comma-separator allows the programmer to tabulate the printing mechanism to fixed tab settings at the end of each print zone.

A completely empty print-list will generate an end-of-line, thereby completing the current line of output. If this line contained no characters, then a blank line results.

A print line on a typical terminal might be divided into five print zones of fifteen print positions each.

15. INPUT STATEMENT

15.1 General Description

Input-statements provide for user interaction with a running program by allowing variables to be assigned values that are supplied by a user. The input-statement enables the entry of mixed string and numeric data, with data items being separated by commas. The general syntactic form of the input-statement is

INPUT variable, ..., variable

15.2 Syntax

1. input-statement = INPUT variable-list
2. variable-list = variable (comma variable)*
3. input-prompt = [implementation-defined]
4. input-reply = input-list end-of-line
5. input-list = padded-datum (comma padded-datum)*
6. padded-datum = space* datum space*
7. datum = quoted-string / unquoted-string

15.3 Examples

```
INPUT X          INPUT X, A$, Y(2)      INPUT A, B, C
3.14159          2,SMITH,-3             25,0,-15
```

15.4 Semantics

An input-statement causes the variables in the variable-list to be assigned, in order, values from the input-reply. In the interactive mode, the user of the program is informed of the need to supply data by the output of an input-prompt. In batch mode, the input-reply is requested from the external source by an implementation-defined means. Execution of the program is suspended until a valid input-reply has been supplied.

The type of each datum in the input-reply shall correspond to the type of the variable to which it is to be assigned; i.e., numeric-constants shall be supplied as input for numeric-variables, and either quoted-strings or unquoted-strings shall be supplied as input for string-variables. If the response to input for a string-variable is an unquoted-string, leading and trailing spaces shall be ignored (see 4).

If the evaluation of a numeric datum causes an underflow, then its value shall be replaced by zero.

Subscript expressions in the variable-list are evaluated after values have been assigned to the variables preceding them (i.e. to the left of them) in the variable-list.

No assignment of values in the input-reply shall take place until the input-reply has been validated with respect to the type of each datum, the number of input items, and the allowable range for each datum.

15.5 Exceptions

- The type of datum does not match the type of the variable to which it is to be assigned (nonfatal, the recommended recovery procedure is to request that the input-reply be re-supplied).
- There is insufficient data in the input-list (nonfatal, the recommended recovery procedure is to request that the input-reply be resupplied).
- There is too much data in the input-list (nonfatal, the recommended recovery procedure is to request that the input-reply be resupplied).

- The evaluation of a numeric datum causes an overflow (nonfatal, the recommended recovery procedure is to request that the input-reply be resupplied).
- A string datum contains too many characters (nonfatal, the recommended recovery procedure is to request that the input-reply be resupplied).

15.6 Remarks

This Standard does not require an implementation to perform any editing of the input-reply, though such editing may be performed by the operating environment.

It is recommended that the input-prompt consists of a question-mark followed by a single space.

This Standard does not require an implementation to output the input-reply.

It is recommended that implementations report an underflow as an exception and allow the input-reply to be resupplied.

16. READ AND RESTORE STATEMENTS

16.1 General Description

The read-statement provides for the assignment of values to variables from a sequence of data created from data-statements (see 17). The restore-statement allows the data in the program to be reread. The general syntactic forms of the read and restore statements are

```
READ variable, ..., variable
RESTORE
```

16.2 Syntax

1. read-statement = READ variable-list
2. restore-statement = RESTORE

16.3 Examples

```
READ X, Y, Z          READ X(1), A$, C
```

16.4 Semantics

The read-statement causes variables in the variable-list to be assigned values, in order, from the sequence of data (see 17). A conceptual pointer is associated with the data sequence. At the initiation of execution of a program, this pointer points to the first datum in the data sequence. Each time a read-statement is executed, each variable in the variable-list in sequence is assigned the value of the datum indicated by the pointer and the pointer is advanced to point beyond that datum.

The restore-statement resets the pointer for the data sequence to the beginning of the sequence, so that the next read-statement executed will read data from the beginning of the sequence once again.

The type of a datum in the data sequence shall correspond to the type of the variable to which it is to be assigned; i.e., numeric-variables require unquoted-strings which are numeric-constants as data and string-variables require quoted-strings or unquoted-strings as data. An unquoted-string which is a valid numeric representation may be assigned to either a string-variable or a numeric-variable by a read-statement.

If the evaluation of a numeric datum causes an underflow, then its value shall be replaced by zero.

Subscript expressions in the variable-list are evaluated after values have been assigned to the variables preceding them (i.e. to the left of them) in the list.

16.5 Exceptions

- The variable-list in a read-statement requires more data than are present in the remainder of the data-sequence (fatal).
- An attempt is made to assign a string datum to a numeric variable (fatal).
- The evaluation of a numeric datum causes an overflow (non-fatal, the recommended recovery procedure is to supply machine infinity with the appropriate sign and continue).
- A string datum contains too many characters (fatal).

16.6 Remarks

It is recommended that implementations report an underflow as exception and continue.

17. DATA STATEMENT

17.1 General Description

The data-statement provides for the creation of a sequence of representations for data elements for use by the read-statement. The general syntactic form of the data-statement is

DATA datum, ..., datum

where each datum is either a numeric constant, a string-constant or an unquoted string.

17.2 Syntax

1. data-statement = DATA data-list
2. data-list = datum (comma datum)*

17.3 Examples

```
DATA 3.14159, PI, 5E-10, ","
```

17.4 Semantics

Data from the totality of data-statements in the program are collected into a single data sequence. The order in which data appear textually in the totality of all data-statements determines the order of the data in the data sequence.

If the execution of a program reaches a line containing a data-statement, then it shall proceed to the next line with no other effect.

17.5 Exceptions

None.

18. ARRAY DECLARATIONS

18.1 General Description

The dimension-statement is used to reserve space for arrays. Unless declared otherwise, all array subscripts shall have a lower bound of zero and an upper bound of ten. Thus the default space allocation reserves space for 11 elements in one-dimensional arrays and 121 elements in two-dimensional arrays. By use of a dimension-statement, the subscript(s) of an array may be declared to have an upper bound other than ten. By use of an option-statement, the subscripts of all arrays may be declared to have a lower bound of one.

The general syntactic form of the dimension-statement is

DIM declaration, ..., declaration

where each declaration has the form

letter (integer)
or letter (integer , integer)

The general syntactic form of the option-statement is

OPTION BASE n

where n is either 0 or 1.

18.2 Syntax

1. dimension-statement = DIM array declaration (comma array-declaration)*
2. array-declaration = numeric-array-name left-parenthesis bounds right-parenthesis
3. bounds = integer (comma integer)?
4. option-statement = OPTION BASE (0/1)

18.3 Examples

```
DIM A (6), B(10,10)
```

18.4 Semantics

Each array-declaration occurring in a dimension-statement declares the array named to be either one or two dimensional according to whether one or two bounds are listed for the array. In addition, the bounds specify the maximum values that subscript expressions for the array can have.

The declaration for an array, if present at all, shall occur in a lower numbered line than any reference to an element of that

array. Arrays that are not declared in any dimension-statement are declared implicitly to be one or two dimensional according to their use in the program, and to have subscripts with a maximum value of ten (see 7).

The option-statement declares the minimum value for all array subscripts; if no option-statement occurs in a program, this minimum is zero. An option-statement, if present at all, must occur in a lower numbered line than any dimension-statement or any reference to an element of an array. If an option-statement specifies that the lower bound for array subscripts is one, then no dimension-statement in the program may specify an upper bound of zero. A program may contain at most one option-statement.

If the execution of a program reaches a line containing a dimension-statement or an option-statement, then it shall proceed to the next line with no other effect.

An array can be explicitly dimensioned only once.

18.5 Exceptions

None.

19. REMARK STATEMENT

19.1 General Description

The remark-statement allows program annotation.

19.2 Syntax

1. remark-statement = REM remark-string

19.3 Examples

REM FINAL CHECK

19.4 Semantics

If the execution of a program reaches a line containing a remark-statement, then it shall proceed to the next line with no other effect.

19.5 Exceptions

None.

20. RANDOMIZE STATEMENT

20.1 General Description

The randomize-statement overrides the implementation-predefined sequence of pseudo-random numbers as values for the RND function, allowing different (and unpredictable) sequences each time a given program is executed.

20.2 Syntax

1. randomize-statement = RANDOMIZE

20.3 Examples

RANDOMIZE

20.4 Semantics

Execution of the randomize-statement shall generate a new unpredictable starting point for the list of pseudo-random numbers used by the RND function (see 9).

20.5 Exceptions

None.

20.6 Remarks

In the case of implementations which do not have access to a randomizing device such as a real-time clock, the randomize-statement may be implemented by means of an interaction with the user.

NAME	GRAPHIC
Space	
Exclamation-mark	!
Quotation-mark	"
Number-sign	#
Dollar-sign	$
Percent-sign	%
Ampersand	&
Apostrophe	'
Left-parenthesis	(
Right-parenthesis	)
Asterisk	*
Plus-sign	+
Comma	,
Minus-sign	-
Full-stop	.
Solidus	/
Digits	0 - 9
Colon	:
Semi-colon	;
Less-than-sign	<
Equals-sign	=
Greater-than-sign	>
Question-mark	?
Letters	A - Z
Circumflex-accent	^
Underline	_

TABLE 1

b_4	b_3	b_2	b_1		b_7 0	0	0	0	1	1	1	1
					b_6 0	0	1	1	0	0	1	1
					b_5 0	1	0	1	0	1	0	1
					0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	TC$_7$ (DLE)	SP	0	@	P	`	p
0	0	0	1	1	TC$_1$ (SOH)	DC$_1$	!	1	A	Q	a	q
0	0	1	0	2	TC$_2$ (STX)	DC$_2$	"	2	B	R	b	r
0	0	1	1	3	TC$_3$ (ETX)	DC$_3$	#	3	C	S	c	s
0	1	0	0	4	TC$_4$ (EOT)	DC$_4$	¤	4	D	T	d	t
0	1	0	1	5	TC$_5$ (ENQ)	TC$_8$ (NAK)	%	5	E	U	e	u
0	1	1	0	6	TC$_6$ (ACK)	TC$_9$ (SYN)	&	6	F	V	f	v
0	1	1	1	7	BEL	TC$_{10}$ (ETB)	'	7	G	W	g	w
1	0	0	0	8	FE$_0$ (BS)	CAN	(	8	H	X	h	x
1	0	0	1	9	FE$_1$ (HT)	EM	)	9	I	Y	i	y
1	0	1	0	10	FE$_2$ (LF)	SUB	*	:	J	Z	j	z
1	0	1	1	11	FE$_3$ (VT)	ESC	+	;	K	[	k	{
1	1	0	0	12	FE$_4$ (FF)	IS$_4$ (FS)	,	<	L	\	l	|
1	1	0	1	13	FE$_5$ (CR)	IS$_3$ (GS)	-	=	M	]	m	}
1	1	1	0	14	SO	IS$_2$ (RS)	.	>	N	^	n	~
1	1	1	1	15	SI	IS$_1$ (US)	/	?	O	_	o	DEL

TABLE 2

NOTE: *In the 7-bit and in the 8-bit code tables two characters are allocated to pos. 2/4, namely $ and ¤. In any version of the codes a single character is to be allocated to this position. The character of the 7-bit or of the 8-bit coded character set, which corresponds to the character $ of the Minimal BASIC character set is either $ or ¤ (¤ in the International Reference Version).*

The same applies to pos. 2/3 for the characters £ and #, the latter being the character of the International Reference Version.

APPENDIX 1

Organization of the Standard

This Standard is organized into a number of sections, each of which covers a particular feature of BASIC. Sections 4 to 20 are divided into sub-sections, as follows.

Sub-section 1. General Description

This sub-section briefly describes the features of BASIC to be treated and indicates the general syntactic form of these features.

Sub-section 2. Syntax

The exact syntax of features of the language is described in a modified context-free grammar or Backus-Naur Form. The details of this method of syntax specification are described in Appendix 2.

In order to keep the syntax reasonably simple the syntax specification will allow it to describe some constructions which, strictly speaking, are not legal according to this Standard, e.g. it will allow the generation of the statement

```
100 LET X = A(1) + A(1,2)
```

in which the array A occurs with differing numbers of subscripts. Rather than ruling such constructions out by a more complicated syntax, this Standard shall instead rule them out in the semantics.

Sub-section 3. Examples

A short list of valid examples that can be generated by certain of the syntax equations in sub-section 2 is given.

Sub-section 4. Semantics

The semantic rules in this Standard serve two purposes. First, they rule out certain constructions which are permitted by the syntax, but which have no valid meaning according to this Standard. Second, they assign a meaning to the remaining constructions.

Sub-section 5. Exceptions

An exception occurs when an implementation recognizes that a program may not perform or is not performing in accordance with this Standard. All exceptions described in this section shall be reported unless some mechanism is provided in an enhancement to this Standard that has been invoked by the user to handle exceptions.

Where indicated, certain exceptions may be handled by the specified procedures; if no procedure is given, or if restrictions imposed by

the hardware or the operating environment make it impossible to follow the given procedures, then the exception must be handled by terminating the program. Enhancements to this Standard may describe mechanisms for controlling the manner in which exceptions are reported and handled, but no such mechanisms are specified in this Standard.

This Standard does not specify an order in which exceptions shall be detected or processed.

Sub-section 6. Remarks

This sub-section contains remarks which point out certain features of this Standard as well as remarks which make recommendations concerning the implementation of a BASIC language processor in an operating environment.

APPENDIX 2

Method of Syntax Specification

The syntax, through a series of rewriting rules known as "productions", defines syntactic objects of various types, such as "program" or "expression", and describes which strings of symbols are objects of these types.

In the syntax, upper-case letters, digits, and (possibly hyphenated) lower-case words are used as "metanames", i.e. as names of syntactic objects. Most of these metanames are defined by rewriting rules in terms of other metanames. In order that this process terminate, certain metanames are designated as "terminal" metanames, and rewriting rules for them are not included in the syntax. All terminal metanames occur for the first time and are defined in Section 4. It should be noted in particular that all upper-case letters are terminal metanames which denote themselves.

We illustrate further details of the syntax by considering some examples. In Section 12 we find the production

gosub-statement = GO space* SUB line-number

which indicates that a "gosub-statement" consists of the letters G, O, any number of spaces, S, U, and B followed by a line number.

What is a "line-number"? In Section 5, the production

line-number = digit digit? digit? digit?

indicates that a "line-number" is a "digit" followed by up to three other "digits" (the question mark is a syntactic operator indicating that the object it follows may or may not be present).

What is a "digit"? In Section 4, the production

digit = 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9

indicates that a "digit" is either a "0", a "1", ... or a "9" (the solidus is a syntactic operator meaning "or" and is used to indicate that a metaname can be rewritten in one of several ways). Since the digits are terminal metanames (i.e. they do not occur on the left-hand side of any production), our decipherment of the syntax for the "gosub-statement" comes to an end. The semantics in Section 4 identify the digits in terms of the characters they represent.

An asterisk is a syntactic operator like the question-mark, and it indicates that the object it follows may occur any number of times, including zero times, in the production.

For example

integer = digit digit*

indicates that an "integer" is a "digit" followed by an arbitrary

number of other "digits".

Parentheses may be used to group sequences of metanames together. For example

```
variable-list = variable (comma variable)*
```

defines a "variable-list" to consist of a "variable" followed by an arbitrary number of other "variables" separated by "commas".

When several syntactic operators occur in the same production, the operators "?" and "*" take precedence over the operator "/".

Spaces in the syntax are used to separate hyphenated lower-case words from each other. Special conventions are observed regarding spaces in BASIC programs (see Section 5). The syntax as described generates programs which contain no spaces other than those occurring in remarks, in certain string constants, or where the presence of a space is explicitly indicated by the metaname "space".

Additional spaces may be inserted to improve readability provided that the restrictions imposed in Section 5 are observed.

APPENDIX 3

Conformance

There are two aspects of conformance to this language Standard : conformance by a program written in the language, and conformance by an implementation which processes such programs.

A program is said to conform to this Standard only when

- each statement contained therein is a syntactically valid instance of a statement specified in this Standard,
- each statement has an explicitly valid meaning specified herein, and
- the totality of statements compose an instance of a valid program which has an explicitly valid meaning specified herein.

An implementation is said to conform to this Standard only when

- it accepts and processes programs conforming to this Standard,
- it reports reasons for rejecting any program which does not conform to this Standard,
- it interprets errors and exceptional circumstances according to the specifications of this Standard,
- its interpretation of the semantics of each statement of a standard-conforming program conforms to the specification in this Standard,
- its interpretation of the semantics of a standard-conforming program as a whole conforms to the specifications in this Standard,
- it accepts as input, manipulates, and can generate as output numbers of at least the precision and range specified in this Standard, and
- it is accompanied by a reference manual which clearly defines the actions taken in regard to features which are called "undefined" or "implementation-defined" in this Standard.

This Standard does not include requirements for reporting specific syntactic errors in the text of a program. Implementations conforming to this Standard may accept programs written in an enhanced language without having to report all constructs not conforming to this Standard. However, whenever a statement or other program element does not conform to the syntactic rules given herein, either an error shall be reported or the statement or other program element shall have an implementation-defined meaning.

APPENDIX 4

Implementation-defined Features

A number of the features defined in this Standard have been left for definition by the implementor. However, this will not affect portability, provided that the limits recommended in the various sections are respected. The way these features are implemented shall be defined in the user- or system-manual of the specific implementation.

The following is a list of implementation-defined features:

- accuracy of evaluation of numeric expressions (see 8)
- end-of-line (see 5, 14 and 15)
- exrad-width for printing numeric representations (see 14)
- initial value of variables (see 7)
- input-prompt (see 15)
- longest string that can be retained (see 11)
- value of machine infinitesimal (see 6)
- value of machine infinity (see 6)
- margin for output lines (see 14)
- precision for numeric values (see 6)
- print-zone length (see 14)
- pseudo-random number sequence (see 9 and 20)
- significance width for printing numeric representations (see 15)
- means of requesting the input-reply in batch mode (see 15)